U.S. Department of Commerce
Patent and Trademark Office
Assistant Commissioner of Patents
Technology Center 3600
2451 Crystal Drive, Arlington VA

FAX COVER SHEET

To: Ms Katherine Voisinet		**From:** Jim Zurita	
Fax: 703-205-8050		**Art Unit:** 3625	
Serial No.: 09-372750		**Date:** 4/28/03	
CC:		**Phone No.:** 703-605-4966	

☐ **Urgent** ☐ **For Review** ☐ **Please Comment** ☐ **Please Reply** ☐ **Per Your Request**

- **Comments:**

as discussed plz call to review - thx.

Number of Pages 34, including this page.

sheets of paper are pulled in by the scanner and scanned as they pass over a stationary scanning mechanism. Sheet-fed scanners allow for automatic scanning of multiple-sheet documents. *See also* scanner. *Compare* drum scanner, flatbed scanner, handheld scanner.

sheet feeder \shēt´ fē`dər\ *n.* A device that accepts a stack of paper and feeds it to a printer one page at a time.

shelfware \shelf´wâr\ *n.* Software that has been unsold or unused for a long time, and so has remained on a retailer's or user's shelf.

MS

shell[1] \shel\ *n.* A piece of software, usually a separate program, that provides direct communication between the user and the operating system. Examples of shells are Macintosh Finder and the MS-DOS command interface program COMMAND.COM. ~~*See also*~~ Bourne shell, C shell, Finder, Korn shell. *Compare* kernel. Unix

~~**shell**[2] \shel\ *vb. See* shell out.~~

shell account \shel´ ə-kount`\ *n.* A computer service that permits a user to enter operating-system commands on the service provider's system through a command-line interface (usually one of the UNIX shells) rather than having to access the Internet through a graphical user interface. Shell accounts can provide Internet access through character-based tools, such as Lynx for browsing the World Wide Web. *See also* shell[1].

shell archive \shel´ är`kīv\ *n.* In UNIX and GNU, a collection of compressed files that has been prepared for transmission by an e-mail service using the shar command.

shell out \shel out´\ *vb.* To obtain temporary access to the operating-system shell without having to shut down the current application and return to that application after performing the desired shell function. Many UNIX programs allow the user to shell out; the user can do the same in windowing environments by switching to the main system window.

shell script \shel´ skript\ *n.* A script executed by the command interpreter (shell) of an operating system. The term generally refers to scripts executed by the Bourne, C, and Korn shells on UNIX platforms. *Also called* batch file. *See also* batch file, script, shell[1].

Shell sort \shel´ sōrt\ *n.* A programming algorithm used for ordering data. Named after its inventor, Donald Shell, it is faster than the bubble sort and the insertion sort. *See also* algorithm. *Compare* bubble sort, insertion sort.

shift \shift\ *vb.* In programming, to move the bit values one position to the left or right in a register or memory location. *See also* end-around shift. *Compare* rotate (definition 2).

Shift+click or **Shift click** \shift-klik´\ *vb.* To click the mouse button while holding down the Shift key. Shift+clicking performs different operations in different applications, but its most common use in Windows is to allow users to select multiple items in a list, for example, to select a number of files for deletion or copying.

Shift key \shift´ kē\ *n.* A keyboard key that, when pressed in combination with another key, gives that key an alternative meaning; for example, producing an uppercase character when a letter key is pressed. The Shift key is also used in various key combinations to create nonstandard characters or to perform special operations. The term is adapted from usage in relation to manual typewriters, in which the key physically shifted the carriage to print an alternative character. *See also* Caps Lock key.

URL

Shift-PrtSc \shift`print´skrēn\ *n. See* Print Screen key.

shift register \shift´ rej`ə-stər\ *n.* A circuit in which all bits are shifted one position at each clock cycle. It can be either linear (a bit is inserted at one end and "lost" at the other during each cycle) or it can be *cyclic* or *looped* (the "lost" bit is inserted back at the beginning). *See also* register, shift.

Shockwave \shok´wāv\ *n.* A format for multimedia audio and video services within HTML documents, created by Macromedia, which markets a family of Shockwave servers and plug-in programs for Web browsers. *See also* HTML.

short card \shōrt´ kärd\ *n.* A printed circuit board that is half as long as a standard-size circuit board. See the illustration on the next page. *Also called* half-card. *See also* printed circuit board.

short-circuit evaluation \shōrt`sər´kət i-val-yōō-ā`shən\ *n.* A form of expression evaluation that

MS dictionary
'97

Programming Windows® 95

Charles Petzold

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Petzold, Charles, 1953–
Programming Windows 95 / Charles Petzold. -- 4th ed.
p. cm.
Includes index.
ISBN 1-55615-676-6
1. Microsoft Windows (Computer file) 2. Operating systems (Computers) I. Title.
QA76.76.O63P533 1996
005.265--dc20 95-49555
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QMQM 1 0 9 8 7 6

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (206) 936-7329.

Acquisitions Editor: Eric Stroo
Project Editor: Jack Litewka
Technical Editor: Marc Young

4

Chapter 20

What's This Thing Called OLE?

OLE is a set of standards for building connectable component software. One OLE standard is the Component Object Model (COM) specification, a blueprint for the binary connections between components. Another defining element of OLE is a set of dynamic link libraries that are part of Windows 95 and Windows NT. Microsoft has arranged for third parties—including Digital, Software AG, and Bristol Technologies—to port some of the technologies from these libraries to other operating system platforms. But the set of services provided by OLE is not static. Just as Microsoft has continually enhanced and extended the Windows operating system, so too will Microsoft continue to evolve OLE to accommodate a broader range of application integration needs. As of this writing, for example, Network OLE is slated to appear as part of Windows NT version 4.0; it will enable component connections across network boundaries.

OLE allows a degree of integration between software modules that previously had required proprietary knowledge and therefore represents the first step in creating a world of interchangeable software components. In such a world, a user would be able to combine a word processing edit window from one vendor with a spelling checker from a second vendor to a print preview component from yet a third software provider.

At present, just a few categories of standard OLE connections have been established. And yet, the architecture of OLE not only allows for more categories, but makes new families of connections inevitable. Prior to Windows 95, the four most common categories of OLE connections were compound document support, OLE Automation, OLE Controls, and the Extended Messaging Applications Programming Interface (Extended MAPI). Windows

6

95 introduced a new family of OLE components, *shell extensions*, for creating tight integration between application software and the Windows 95 desktop. Another OLE-aware component introduced with Windows 95 is the rich edit control, a dialog box control that provides a simple compound document container.

A compound document container creates compound documents, which can hold data from many different, unrelated applications. Containers communicate with "object server" applications to negotiate the two-way movement of data between a server and a compound document. An example of a compound document is when a portion of an Excel spreadsheet is embedded in a Microsoft Word for Windows document. In this example, the Excel spreadsheet data is an *embedded object*, and Word for Windows provides a *compound document container.* This example doesn't show OLE at its best, however, since it's easy to suspect collusion when two products from the same company cooperate closely. That OLE's compound document support is truly universal is demonstrated when programs from many vendors—Adobe PageMaker or Micrografx Designer, for example—provide the same compound document support as Microsoft Word. Each of the container applications can hold Excel spreadsheet objects, Corel Draw drawing objects, Visio diagrams, or data objects from any OLE-compliant object server.

Automation provides a mechanism for defining a set of macro primitives. Macro primitives consist of methods (another term for function calls) and properties (that is, data elements that can be read from or written to or that can be read-only or write-only). The term "automation object" refers to an OLE component that provides a macro primitive. An "automation controller" manipulates the methods and properties of an automation object. Through the OLE-defined standards for automation, programming environments such as Microsoft's Visual Basic, Borland's Delphi, or PowerSoft's Power Builder let you create automation controllers to provide centralized control of work distributed among specialized applications.

OLE Controls are a third type of standard OLE component; they are like dialog box controls in that they are essentially special purpose child windows. Instead of residing in dialog boxes, however, OLE controls reside within *OLE control containers.* OLE controls are automation objects and export their own macro primitives. OLE controls are like compound document objects in that they can save state information in a file created and managed by their container application. OLE controls recognize *ambient properties*, attributes like background color and current font, which allow them to visually blend into their container. OLE controls have the ability to send event notifications to the control container, in that way, they are like Visual Basic controls, which send events to trigger responses. In fact, Microsoft has publicly announced that Visual Basic controls (VBXs) will not be supported in the 32-bit world and is encouraging all VBX developers to upgrade their Win16 VBXs to Win32 OLE controls.

The common element among all standard OLE connections is that they are glued together with the OLE Component Object Model (COM). COM is an object-oriented specification for defining *interfaces*, the contact points between components. COM provides the foundation on which all OLE features are built.

6

the way that OLE and Win32 handle error codes is that most OLE functions return an HRESULT value directly, whereas Win32 functions don't directly return an error code. For Win32 functions, such error codes are available by calling *GetLastError*. This function returns the current error code for the last function that failed on the current thread. Some Win32 routines clear the error code—to set it to a known state before calling other Win32 routines that might fail—so an immediate check on this code is necessary before calling other Win32 functions.

So far, we've touched on two basic OLE programming points: First, to use OLE as a component you must first connect to the OLE libraries. Second, proper interpretation of the HRESULT return code is necessary to distinguish successful connections from unsuccessful ones. Once connected to the OLE libraries, a module has two available choices: connect to an OLE interface (that is, become a client to an OLE interface), or make an OLE interface available for others to use (that is, become a server for an OLE interface). Most OLE-aware programs do both. To help you decide which you should consider first, the next section introduces the fundamentals of OLE interfaces. OLE interfaces are created by following the *Component Object Model (COM)* specification and therefore are also sometimes called *COM Interfaces*.

Component Object Model (COM) Interfaces

The Component Object Model (COM) is a specification for building an OLE-aware component. COM provides a definition for the fundamental bindings required between interface provider and interface user. All OLE interfaces—whether for compound documents, OLE Automation, OLE Controls, Windows 95 shell extensions, or Extended MAPI—are built according to the COM specification.

COM interfaces provide a single solution to a problem that previously had been addressed by many solutions—namely, connecting software components running in different processes or on different computers. The challenge, simply stated, is to efficiently connect components separated by an address space boundary, a network boundary, or no boundary at all. (As of this writing, Network OLE has not been released but is expected shortly after this book goes to press.) OLE interfaces provide for both inter-process and inter-machine cases while also efficiently handling connections between components running in the *same* address space. A single solution to all three connectivity cases means that developers can ignore differences based solely on component location and instead can focus on the task at hand. This is analogous to the way telephones are used: people communicate using a single mechanism—the telephone handset—whether in different rooms of the same building, on different sides of the same continent, or on different hemispheres of the same planet.

As with all good designs, interfaces combine several things together. Interfaces are at the same time a contract for services, the binary connection between the provider for a service and the user of that service, and a mechanism for supporting out-of-process connections (that is, both inter-process and inter-machine connections). What ties these aspects together is a set of core functions that all interfaces share. Each of these points deserves closer consideration.

An interface is a contract for services

An interface is a contract for services provided by one component (the "server") and used by another (the "client"). As such, the only code associated with an interface definition consists of a set of functions prototypes—function return values, function names, and function parameters. The contract specifies *what* services are provided, but not *how* they are to be provided. The details of the implementation are hidden from view—that is, *encapsulated*—within the service provider.

For example, here are the function prototypes for *IMalloc*, OLE's memory allocation interface:

Function Prototype	***Description***
void *Alloc (ULONG cb) ;	Allocate *cb* bytes.
void *Realloc (void *pv, ULONG cb) ;	Resize *pv* memory to *cb* bytes.
void Free (void *pv) ;	Free memory referenced by *pv*.
ULONG GetSize (void *pv) ;	Query memory size.
int DidAlloc (void *pv) ;	Allocated by this allocator?
void HeapMinimize (void) ;	Defragment memory.

The services provided by such an allocator—the "what"—are obvious to any programmer experienced with dynamic memory allocation. But the *implementation*—that is, *how* the requests get satisfied—is not specified. An *IMalloc* implementor on Windows 95 could use any available Win32 mechanism including the page allocator (*VirtualAlloc*) for large objects, the heap allocator (*HeapAlloc*) for small objects, or a Win16 API-compatible allocator (*GlobalAlloc* or *LocalAlloc*) for an allocator compatible with both the Win16 and Win32 APIs. An implementor could even call C run-time allocation functions to create a portable *IMalloc* independent of the Win32 API.

One point worth mentioning about *IMalloc* concerns the naming standard for interfaces. Following the Hungarian notation naming style, interface names have an "I" prefix: for example, *IStorage*, *IOleObject*, and *IDataObject*. This is similar to the use of the "C" prefix with Microsoft MFC library class names (*CWinApp*, *CFrameWnd*) or the "T" prefix with Borland OWL class names (*TApplication*, *TWindow*).

While the details of how an interface gets implemented are hidden from view, the binary contact point between a client and a server is quite visible and clearly defined.

8

An interface defines a binary contact point

At its core, the binary implementation of an interface is a function pointer array. A server for an interface like *IMalloc* implements individual functions and builds an array of function pointers. A client expects to access server functions via such an array.

The specific connection that servers provide to clients is a *pointer to an array of function pointers.* In C, array syntax provides one way to create a function pointer array. An *IMalloc* server could allocate an array of function pointers as follows:

```
// Define generic function pointer type
typedef (* PFNINTERFACE) () ;

// Allocate an array of function pointers
PFNINTERFACE allocator[6] = { Alloc, Realloc, Free,
                              GetSize, DidAlloc,
                              HeapMinimize } ;
```

and then return the critical connection—the pointer to the array of function pointers—to any client that calls:

```
PFNINTERFACE *FetchAllocator ()
     {
     return &allocator ;
     }
```

The problem with array syntax is that calls to *IMalloc* functions are clumsy and error prone. The return type must be cast, no parameter checking can take place, and an array index replaces a function name. A call to *Alloc* requesting 20 bytes, for example, looks like this:

```
PFNINTERFACE *pAlloc = FetchAllocator () ;
void *pData = (void *) pAlloc[0] (20) ;
```

With parameter type checking disabled, calls like the following don't generate any warning or error messages:

```
// Error: Wrong parameter type!!
pData = (void *) pMalloc[0] ("20 bytes") ;

// Error: Wrong number of parameters!!
pData = (void *) pMalloc[0] (20, 30, 40, 50, 60) ;
```

A better approach involves using a structure containing function pointers. Taking this approach involves declarations like the following:

```
// Specifically-typed function pointer types
typedef void * (* PFNALLOC) (ULONG) ;
typedef void * (* PFNREALLOC) (void *, ULONG) ;
typedef void   (* PFNFREE) (void *) ;
```

Role of the Registry

The registry is a central depository for systemwide persistent state in Windows 95 and Windows NT. The registry was first introduced in Windows 3.1 to publish OLE class details, default extensions for the system shell, and a few odd DDE commands. In Windows 95 and Windows NT, the registry is the replacement for state that used to reside in .INI files, such as currently installed hardware, control panel options, and settings of user preferences for application software. While significant differences exist between how each operating system uses the registry, OLE-related registry entries are the same in both systems.

The registry is structured as a hierarchy of keys, subkeys, and values. On Windows 95, the defined root keys include HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_CONFIG, and HKEY_DYN_DATA. (Windows NT provides only the first four.) Data about OLE components reside in HKEY_CLASSES_ROOT. (As a side note, the true location of this hierarchy is HKEY_LOCAL_MACHINE\SOFTWARE\Classes, but for compatibility with Windows 3.1 OLE applications, OLE classes were given their own surrogate root node.)

The root keys of HKEY_CLASSES_ROOT consist of three types of entries: file extensions, class names, and system entries. File extension entries start with a period (.) to associate a file with a compound document server (for example, ".vsd" to Visio Corporation's Visio). Class names provide human-readable identifiers for OLE component classes, for which two uses currently exist. The first use for class names is to support OLE 1.0 compound document servers; this use is identified with class names like "Visio.Drawing.4". The second use for class names is OLE automation. Automation macro primitives create OLE automation objects by name—in this context a class name is called a "Program ID" or just "ProgID." For example, using the "Visio.Application" ProgID, a Visual Basic program can create and manipulate Visio objects via automation methods and properties.

There are three kinds of system registry entries, each of which are the roots of their own hierarchies: *TypeLib*, *Interface*, and CLSID. The *TypeLib* hierarchy identifies the location of currently installed type libraries, which are databases that describe the contents of OLE components. Used extensively for automation support, a type library describes the function prototypes for all supported interfaces and also includes references to help files so that development tools can bring up the appropriate help page to help macro programmers make proper use of automation servers.

The *Interface* hierarchy contains a list—sorted by interface ID—of all interfaces installed on a system. It provides the human-readable name for the interface (*IUnknown*, *IMalloc*, and so forth) and details about each interface (the number of functions and the base class for each interface).

The final hierarchy, the CLSID hierarchy, provides details of all currently installed (public) OLE components. A CLSID is a class identifier. Like interface IDs (IID and REFIID

10

Programming Windows NT 4

UNLEASHED

Mickey Williams

David Hamilton

SAMS PUBLISHING

201 West 103rd Street
Indianapolis, IN 46290

As always, for René, Alexandria, and Mackenzie—Mickey Williams

International Standard Book Number: 0-672-30905-x

Library of Congress Catalog Card Number: 96-67203

99 98 97 96 4 3 2

Interpretation of the printing code: The rightmost double-digit number is the year of the book's printing; the rightmost single-digit, the number of the book's printing. For example, a printing code of 96-1 shows that the first printing of the book occurred in 1996.

Composed in AGaramond and MCPdigital by Macmillan Computer Publishing

Printed in the United States of America

Trademarks

President and Publisher *Richard K. Swadley*

Team Leader *Greg Wiegand*

Managing Editor *Cindy Morrow*

Director of Marketing *John Pierce*

Assistant Marketing Managers *Kristina Perry*
Rachel Wolfe

Acquisitions Editors
Christopher Denny
Brad Jones

Development Editor
Anthony Amico

Software Development Specialist
Steve Straiger

Production Editor
Johnna L. VanHoose

Copy Editors
Fran Blauw, Howard Jones

Technical Reviewers
Robert Bogue
Donald Doherty

Editorial Coordinator
Bill Whitmer

Technical Edit Coordinator
Lynette Quinn

Resource Coordinator
Deborah Frisby

Formatter
Frank Sinclair

Editorial Assistants
Carol Ackerman, Andi Richter, Rhonda Tinch-Mize

Cover Designer
Tim Amrhein

Book Designer
Gary Adair

Copy Writer
Peter Fuller

Production Team Supervisor
Brad Chinn

Production
Carol Bowers, Paula Lowell, Dana Rhodes, Laura A. Smith, Susan Van Ness, Mark Walchle

Indexer
Johnna VanHoose

12

13

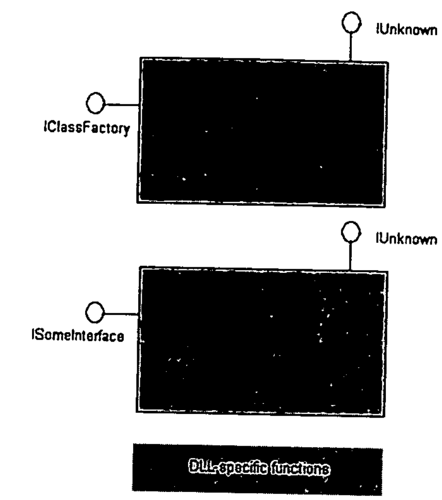

FIGURE 17.15.
Contents of a typical COM custom component server.

Looking at an OLE Example

As an example of how a simple custom component server is implemented, I have created a sample project on the accompanying CD-ROM. The CppExt project is a Windows NT shell extension that creates a custom context menu after you right-click a C++ file in the shell.

Using the Shell Extension Mechanism

The new shell user interface released with Windows NT 4.0 uses COM to support user extensions to the shell. By creating and registering in-process servers supporting the shell extension interfaces, you can create several types of shell extensions, as shown in Table 17.6.

Table 17.6. Windows NT shell extension types.

Shell Extension	*Function*
Icon handlers	Change the appearance of a file's icon on a per-file object basis. By implementing this interface, for example, you can change the icon displayed for a file object based on its internal state, its age, or any other criteria.
Copy hook handlers	Invoked when a file object is copied, moved, or deleted. By implementing this interface, you can supplement or prevent the operation.

continues

Table 17.6. continued

Shell Extension	*Function*
Context menu extensions	Add items to the context menu displayed after a file object is right-clicked.
Property sheet extensions	Add pages to the property sheet displayed by the shell for a particular type of file object.
Drag-and-drop handlers	Called after a drag-and-drop operation. They are almost identical to context menu extensions.
Drop target handlers	Control the activity that occurs when a file object is dropped after a drag-and-drop operation.
Data object handlers	Supply the file object during drag-and-drop operations.

All shell extensions implement the `IShellExtInit` or the `IPersistFile` interfaces. They also support additional interfaces used to implement their particular service.

Understanding Context Menu Extensions

A context menu extension supports two interfaces:

- **`IShellExtInit`:** An interface implemented by several types of shell extensions. `IShellExtInit` defines one function, `Initialize`, that provides information a shell extension can use to initialize itself.
- **`IContextMenu`:** Implemented exclusively by context menu extensions. `IContextMenu` defines three functions. `QueryContextMenu` requests that the shell extension add items to the context menu. `GetCommandString` is used by the shell to collect string descriptions of the added menu items. `InvokeCommand` signals the shell extension that a new menu item has been selected. Each of these interfaces is discussed later in this chapter.

When a user right-clicks on a file object that is handled by a context menu extension, the Windows NT shell creates an instance of the shell extension, using the mechanism discussed earlier for in-process servers.

Next, the shell uses `QueryInterface` for the `IShellExtInit` interface and calls the `Initialize` function, allowing the context menu to store initialization information. A context menu extension must collect information about the file object in this function.

The shell then queries for the `IContextMenu` interface and calls the `QueryContextMenu` function. The shell extension must add all its menu items to the context menu during this function call. The extension is not allowed to remove items or to make any assumptions about the final

15

configuration of the menu. It is possible for several shell extensions to be registered for a single file type, and there is no way to ensure that your extension is called last (or first, for that matter). The shell also calls the `GetCommandString` function in order to collect strings used as canonical verbs and status bar text.

If a menu item added by a context menu extension is selected by the user, `IContextMenu::InvokeCommand` is called by the shell. This is the function that carries out whatever activity is represented by the menu item.

That's all there is to context menu extensions. In addition to the general-purpose code that must be written for all in-process servers, you really need to implement only four functions for your server.

Creating the Example

As with other examples in this book, you can use the project directly from the accompanying CD-ROM, or you can follow the steps described here to create the project from scratch.

WARNING

If you modify this project, you must not reuse my GUID—use UUIDGEN to create your own CLSID.

Get started with the example by using Developer Studio to create a new dynamic link library project workspace named CppExt. Unlike most AppWizard projects, absolutely no code is written for you; you must enter it yourself (or copy it from the CD-ROM).

Exporting the Required DLL Functions

The first file to create for the CppExt project is the module definition file, provided in Listing 17.12. Enter the source code as it is presented here, save it as CppExt.def, and add it to the CppExt project.

Listing 17.12. The module definition file for CppExt.

```
LIBRARY     CppExt
DESCRIPTION "Shell extension for CPP files"

EXPORTS
    DllCanUnloadNow
    DllGetClassObject
```

Notice that only two functions are exported: `DllCanUnloadNow` and `DllGetClassObject`.

16

Creating the Class Declarations

Two classes are declared in the CppExt.h header file. The `CCppShellExt` class is the actual context menu extension class. The `CClassFactory` class is used to create instances of `CCppShellExt`. My normal practice is to put each C++ class in its own header file, but because class factories are coupled tightly to the classes they create, I make an exception for them.

Save the contents of Listing 17.13 as CppExt.h. There's no need to add it to the CppExt project, because it is included in the main .CPP file.

Listing 17.13. The CppExt.h header file.

```
// The class ID for the CPP file extension
/* d3d834c0-9a3b-11cf-82a0-00608cca2a2a */
DEFINE_GUID( CLSID_CppExtension, 0xd3d834c0, 0x9a3b, 0x11cf, 0x82,
             0xa0, 0x00, 0x60, 0x8c, 0xca, 0x2a, 0x2a );

#define     OFFSET_COUNTLINES    0
//
// A typical class factory, nothing out of the ordinary. Includes
// the IClassFactory and IUnknown interfaces
class CClassFactory : public IClassFactory
{
public:
    CClassFactory();
    ~CClassFactory();

    // IUnknown
    STDMETHODIMP QueryInterface( REFIID riid, LPVOID* ppv );
    STDMETHODIMP_(ULONG) AddRef();
    STDMETHODIMP_(ULONG) Release();

    // IClassFactory
    STDMETHODIMP CreateInstance( LPUNKNOWN punk,
                                 REFIID riid,
                                 LPVOID* ppv );
    STDMETHODIMP LockServer( BOOL f );

protected:
    ULONG   m_cRef;
};
//
// The CCppShellExt class
// As a context menu handler, the CCppShellExt class implements the
// IContextMenu and IShellExtInit interfaces.
class CCppShellExt: public IContextMenu, IShellExtInit
{
    ULONG   m_cRef;
    char    m_szFile[MAX_PATH];
public:
    CCppShellExt();
    ~CCppShellExt();
    //
    // IUnknown
    STDMETHODIMP QueryInterface( REFIID riid, LPVOID* ppv );
```

```
    STDMETHODIMP_(ULONG) AddRef();
    STDMETHODIMP_(ULONG) Release();
    //
    // IContextMenu
    STDMETHODIMP QueryContextMenu( HMENU hMenu,
                                   UINT  nMenuIndex,
                                   UINT  nFirstID,
                                   UINT  nLastID,
                                   UINT  nFlags );
    STDMETHODIMP InvokeCommand( LPCMINVOKECOMMANDINFO pInfo );
    STDMETHODIMP GetCommandString( UINT  nItemID,
                                   UINT  nFlags,
                                   UINT* pReserved,
                                   LPSTR pszName,
                                   UINT  cchMax );
    //
    // IShellExtInit
    STDMETHODIMP Initialize( LPCITEMIDLIST pidl,
                             LPDATAOBJECT  pObj,
                             HKEY          hKeyProgID );
private:
    //
    // Operations
    void DoCountCppFile( LPCMINVOKECOMMANDINFO pInfo );
};
```

There are two sections to the CppExt.h file. The first half of the file is almost identical to any header file that declares a class that implements the `IClassFactory` interface. The `CClassFactory` class implements the `IClassFactory` and `IUnknown` interfaces; this declaration can be copied and pasted into any server module you implement.

The second half of the file declares the `CCppShellExt` class. This class implements three interfaces: `IUnknown`, `IShellExt`, and `IShellExtInit`. With the exception of the `DoCountCppFile` function, this code can be reused for any server that implements a context menu extension.

Implementing the In-Process Server Module

The actual implementation of the in-process server is spread out over the next few pages. All this code is physically located in the CppExt.cpp source file. Separating it into three separate listings, however, helps simplify my explanations.

Listing 17.14 presents the first part of the CppExt.cpp source file and deals with the basic DLL functions performed by all in-process servers.

Listing 17.14. Basic in-process server functions in CppExt.cpp.

```
#define STRICT
#define INC_OLE2        // WIN32, get ole2 from windows.h

#include <windows.h>
#include <windowsx.h>
```

continues

18

Listing 17.14. continued

```
#include <shlobj.h>
#pragma data_seg(".text")
#define INITGUID
#include <initguid.h>
#include <shlguid.h>
#pragma data_seg()

#include "CppExt.h"

LONG    g_cRef = 0;
HANDLE  g_hInst = NULL;

// ------------------------------------------------------------
// General DLL functions
// ------------------------------------------------------------
// Main DLL entry point - stash the module instance handle for use
// later, return TRUE in all cases.
extern "C" int APIENTRY
DllMain( HANDLE hInst, ULONG uReason, LPVOID pRes )
{
    if( uReason == DLL_PROCESS_ATTACH )
        g_hInst = hInst;
    return TRUE;
}
//
// Every InProc server must support DllGetClassObject. The only
// object supported by this server is CLSID_CppExtension, all
// other requests are rejected. For our extension, a ClassFactory
// object is created, and a QIF is performed for the requested
// interface through the ClassFactory.
STDAPI DllGetClassObject(REFCLSID rcid, REFIID riid, LPVOID *ppv)
{
    HRESULT hr;
    *ppv = NULL; // Always clear the "out" parameter
    if( IsEqualCLSID( rcid, CLSID_CppExtension ) == FALSE )
        hr = ResultFromScode( CLASS_E_CLASSNOTAVAILABLE );
    else
    {
        CClassFactory* pFactory = new CClassFactory;
        if( pFactory == NULL )
            hr = ResultFromScode( E_OUTOFMEMORY );
        else
        {
            hr = pFactory->QueryInterface( riid, ppv );
        }
    }
    return hr;
}
//
// DllCanUnloadNow is called by the OS to determine if the inproc
// server can be unloaded. If the global reference count is greater
// than zero, S_FALSE is returned to prevent unloading.
STDAPI DllCanUnloadNow()
{
    HRESULT hr = S_FALSE;
    if( g_cRef == 0 )
```

19

```
        hr = S_OK;
    return hr;
}
```

The `DllGetClassObject` function is very similar to a class factory. This function checks that the requesting CLSID matches `CLSID_CppExtension` and creates an instance of the class factory. The class factory then is queried for the requested interface, and the result is returned to the shell.

Listing 17.15 contains the next set of functions implemented in CppExt.cpp. This listing contains the `CClassFactory`, which implements the `IClassFactory` interface.

Listing 17.15. The class factory implementation from CppExt.cpp.

```
// -----------------------------------------------------------
// IClassFactory implementation
// -----------------------------------------------------------
// The IClassFactory interface is responsible for creating an
// instance of the shell extension. The ctor and dtor increment
// and decrement the DLL's global reference count.
CClassFactory::CClassFactory()
{
    m_cRef = 0L;
    InterlockedIncrement( &g_cRef );
}
CClassFactory::~CClassFactory()
{
    InterlockedDecrement( &g_cRef );
}
// IUnknown interfaces for CClassFactory
STDMETHODIMP CClassFactory::QueryInterface( REFIID riid, LPVOID* ppv )
{
    *ppv = NULL;
    if( IsEqualIID( riid, IID_IUnknown ) == TRUE )
    {
        *ppv = (LPUNKNOWN)this;
        m_cRef++;
        return NOERROR;
    }
    else if( IsEqualIID( riid, IID_IClassFactory ) == TRUE )
    {
        *ppv = (LPCLASSFACTORY)this;
        m_cRef++;
        return NOERROR;
    }
    else
    {
        return ResultFromScode( E_NOINTERFACE );
    }
}
STDMETHODIMP_(ULONG) CClassFactory::AddRef()
{
    return ++m_cRef;
```

continues

W

Listing 17.15. continued

```
}
STDMETHODIMP_(ULONG) CClassFactory::Release()
{
    if( --m_cRef )
        return m_cRef;
    delete this;
    return 0L;
}
// IClassFactory interfaces - CreateInstance and LockServer.
//
// CreateInstance creates a CCppShellExt object, and returns
// the result of QIF on the requested interface.
STDMETHODIMP CClassFactory::CreateInstance( LPUNKNOWN punk,
                                            REFIID riid,
                                            LPVOID* ppv )
{
    *ppv = NULL;
    if( punk != NULL )
        return ResultFromScode( CLASS_E_NOAGGREGATION );

    CCppShellExt* pShellExt = new CCppShellExt;
    if( pShellExt == NULL )
        return ResultFromScode( E_OUTOFMEMORY );

    return pShellExt->QueryInterface( riid, ppv );
}
// Simple implementation of LockServer, this just increments and
// decrements the global reference count.
STDMETHODIMP CClassFactory::LockServer( BOOL f )
{
    if( f )
        InterlockedIncrement( &g_cRef );
    else
        InterlockedDecrement( &g_cRef );
    return NOERROR;
}
```

The constructor for `CClassFactory` sets its internal reference count to and increments the module's global reference count, which represents the number of objects created in the entire module. The destructor decrements this same value. Note that the Win32 `InterlockedIncrement` and `InterlockedDecrement` functions are used to change these values. With Windows NT, it is possible for a single process to run with more than one thread at any given time. You must use these functions to ensure that they are properly incremented and decremented.

In addition to the `IUnknown` interfaces, `CClassFactory` implements the two `IClassFactory` functions: `CreateInstance` and `LockServer`. `CreateInstance` uses the `new` operator to create a new instance of `CppShellExt` and returns the result of `QueryInterface` to the client—in this case, the Windows NT shell.

`LockServer` is used by the shell to ensure that the DLL stays loaded, even if all of its objects are destroyed. For an in-process server, a simple solution is to increment and decrement the global

21

object reference count, depending on the value of the flag passed to `LockServer`. There is a great deal of sample code in other books, and even sample programs available from Microsoft imply that this function need not be implemented. This is not correct; all class factories must implement `LockServer`.

Listing 17.16 contains the last group of functions contained in CppExt.cpp. These are the `CCppShellExt` functions, which implement the `IShellExtInit` and `IContextMenu` interfaces.

Listing 17.16. The shell extension implementation from CppExt.cpp.

```
// -----------------------------------------------------------
// Shell extension implementation
// -----------------------------------------------------------
// There are three interfaces supported by a context menu extension
// - ICcontextMenu, IShellExtInit, and IUnknown. Additionally,there
// is one private member function, DoCppCount.
//
// The ctor and dtor for the CppShellExt class increment and decre-
// ment the global reference count for the DLL. The ctor also
// handles initialization of member variables.
CCppShellExt::CCppShellExt()
{
    m_cRef = 0L;
    m_szFile[0] = '\0';
    InterlockedIncrement( &g_cRef );
}
CCppShellExt::~CCppShellExt()
{
    InterlockedDecrement( &g_cRef );
}
// IUnknown interfaces for the Shell Interfaces
STDMETHODIMP CCppShellExt::QueryInterface( REFIID riid, LPVOID* ppv )
{
    if( IsEqualIID( riid, IID_IUnknown ) == TRUE )
    {
        *ppv = (LPUNKNOWN)(LPCONTEXTMENU)this;
        m_cRef++;
        return NOERROR;
    }
    else if( IsEqualIID( riid, IID_IContextMenu ) == TRUE )
    {
        *ppv = (LPCONTEXTMENU)this;
        m_cRef++;
        return NOERROR;
    }
    else if( IsEqualIID( riid, IID_IShellExtInit ) == TRUE )
    {
        *ppv = (LPSHELLEXTINIT)this;
        m_cRef++;
        return NOERROR;
    }
    else
    {
        *ppv = NULL;
        return ResultFromScode( E_NOINTERFACE );
```

continues

Listing 17.16. continued

```
    }
}
STDMETHODIMP_(ULONG) CCppShellExt::AddRef()
{
    return ++m_cRef;
}
STDMETHODIMP_(ULONG) CCppShellExt::Release()
{
    if (--m_cRef)
        return m_cRef;
    delete this;
    return 0L;
}
// ----------------------------------------------------------
// IShellExtInit interface
// ----------------------------------------------------------
// IShellExtInit only has one function - Initialize is called to
// prepare your shell extension for calls that will be made on
// other interfaces - in this case, through IContextMenu. The main
// point of interest for a context menu is the name of the file
// object receiving the mouse-click, which is collected using
// DragQueryFile.
STDMETHODIMP CCppShellExt::Initialize( LPCITEMIDLIST pidl,
                                       LPDATAOBJECT  pObj,
                                       HKEY          hKeyProgID )
{
    STGMEDIUM   stg;
    FORMATETC   fetc = { CF_HDROP,
                         NULL,
                         DVASPECT_CONTENT,
                         -1,
                         TYMED_HGLOBAL };
    if( pObj == NULL )
        return ResultFromScode( E_FAIL );
    HRESULT hr = pObj->GetData( &fetc, &stg );
    if( FAILED(hr) )
        return ResultFromScode( E_FAIL );

    UINT cFiles = DragQueryFile( (HDROP)stg.hGlobal,
                                 0xFFFFFFFF,
                                 NULL,
                                 0 );
    if( cFiles == 1 )
    {
        DragQueryFile( (HDROP)stg.hGlobal,
                       0,
                       m_szFile,
                       sizeof(m_szFile) );
        hr = NOERROR;
    }
    else
        hr = ResultFromScode( E_FAIL );

    ReleaseStgMedium( &stg );
    return hr;
}
```

23

```
// ---------------------------------------------------------
// IContextMenu interfaces
// ---------------------------------------------------------
//
// QueryContextMenu is called when the shell requests the extension
// to add its menu items to the context menu. It's possible to get
// a NULL menu handle here. Also note that the current (As of this
// writing) Win32 SDK documentation is wrong regarding the nFlags
// parameter. This function should always return the number of new
// items added to the menu.
STDMETHODIMP CCppShellExt::QueryContextMenu( HMENU hMenu,
                                             UINT  nMenuIndex,
                                             UINT  nFirstID,
                                             UINT  nLastID,
                                             UINT  nFlags )
{
    char szMenu[] = "Count Lines and Statements";
    BOOL fAppend = FALSE;
    if( (nFlags & 0x000F) == CMF_NORMAL )
    {
        fAppend = TRUE;
    }
    else if( nFlags & CMF_VERBSONLY )
    {
        fAppend = TRUE;
    }
    else if( nFlags & CMF_EXPLORE )
    {
        fAppend = TRUE;
    }
    if( fAppend && hMenu )
    {
        BOOL f = ::InsertMenu( hMenu,
                      nMenuIndex,
                      MF_STRING | MF_BYPOSITION,
                      nFirstID,
                      szMenu );
        return ResultFromScode( MAKE_SCODE(SEVERITY_SUCCESS,
                                           0,
                                           USHORT(1)) );
    }
    return NOERROR;

}
//
// InvokeCommand is the "Money Shot". This is a notification that
// the user has clicked on one of the selected items in the context
// menu. The name of the file is not passed to you in this function
// since it was passed in the IShellExtInit::Initialize  function.
STDMETHODIMP
CCppShellExt::InvokeCommand( LPCMINVOKECOMMANDINFO pInfo )
{
    if( HIWORD(pInfo->lpVerb) != 0 )
        return ResultFromScode( E_FAIL );

    if( LOWORD(pInfo->lpVerb) > OFFSET_COUNTLINES )
        return ResultFromScode( E_INVALIDARG );
```

continues

Listing 17.16. continued

```
    if( LOWORD(pInfo->lpVerb) == OFFSET_COUNTLINES )
        DoCountCppFile( pInfo );
    return NOERROR;
}
//
// GetCommandString is called by the shell to retrieve a string
// associated with a new menu item.
STDMETHODIMP CCppShellExt::GetCommandString( UINT  nItemID,
                                             UINT  nFlags,
                                             UINT* pReserved,
                                             LPSTR pszName,
                                             UINT  cchMax )
{
    if( nItemID == OFFSET_COUNTLINES )
    {
        switch( nFlags )
        {
        case GCS_HELPTEXT:
            lstrcpy( pszName,
                     "Counts lines and semicolons in a file");
            return NOERROR;
            break;
        case GCS_VALIDATE:
            return NOERROR;
            break;
        case GCS_VERB:
            lstrcpy( pszName, "Count" );
            break;
        }
    }
    return ResultFromScode(E_INVALIDARG);
}
//
// DoCountCppFile opens the file which is located under the mouse
// click. The name of this file was passed to us in the Initialize
// member function that is part of the IShellExtInit interface. The
// file name was stored in m_szFile. This function opens the file
// and counts the number of newlines and semicolons in the file.
void CCppShellExt::DoCountCppFile( LPCMINVOKECOMMANDINFO pInfo )
{
    HANDLE  hFile = CreateFile( m_szFile,
                                GENERIC_READ,
                                FILE_SHARE_READ,
                                NULL,
                                OPEN_EXISTING,
                                FILE_ATTRIBUTE_COMPRESSED,
                                NULL );
    if( hFile == INVALID_HANDLE_VALUE )
    {
        ::MessageBox( pInfo->hwnd,
                      m_szFile,
                      "Can't open file",
                      MB_ICONHAND );
        return;
    }
    BOOL  fRead;
```

25

```
    DWORD dwRead;
    DWORD cSemi = 0L;
    DWORD cLines = 0L;
    while(1)
    {
        char rgBuffer[1024];
        fRead = ReadFile( hFile,
                          rgBuffer,
                          sizeof(rgBuffer),
                          &dwRead,
                          NULL );
        if( fRead == FALSE || dwRead == 0 )
            break;
        for( DWORD dw = 0; dw < dwRead; dw++ )
        {
            if( rgBuffer[dw] == ';' )
                cSemi++;
            else if( rgBuffer[dw] == '\n' )
                cLines++;
        }
    }
    TCHAR szMsg[80];
    TCHAR szSemi[] = "Total Semicolons = ";
    TCHAR szLines[] = "Total Lines = ";
    wsprintf( szMsg,"%s%ld\n%s%ld",
              (LPCTSTR)szSemi,
              (DWORD)cSemi,
              (LPCTSTR)szLines,
              (DWORD)cLines );
    ::MessageBox( pInfo->hwnd, szMsg, "C++ File", MB_ICONINFORMATION );
    CloseHandle( hFile );
}
```

Save the contents from Listings 17.14, 17.15, and 17.16 as CppExt.cpp and add this file to the CppExt project. After compiling the CppExt project, you are ready to register the extension and copy the DLL into the Windows NT system directory, which is covered in the next section.

Registering the Shell Extension

Like all in-process servers, a shell extension must be registered before it is used. All in-process servers must be registered in the `HKEY_CLASSES_ROOT\CLSID` key. The simplest way to implement these changes is to create a Registry file, which will be merged into the system Registry.

Create a new key for the CLSID used by the shell extension and give it a string value with an easy-to-read name—in this case, `C++ Line Counter`. Under this key, add an `InProcServer32` key that marks this class as an in-process server. The value associated with this key is the name of the DLL that implements the server—in this case, `CppExt.dll`. You also must add a `ThreadingModel` key, which always is set to `Apartment`, as shown in Listing 17.17.

26

Listing 17.17. Registry file entries required for a thread-safe in-process server.

```
[HKEY_CLASSES_ROOT\CLSID\<GUID>]
   @="C++ Line Counter"
[HKEY_CLASSES_ROOT\CLSID\<GUID>\InProcServer32]
   @="CppExt.dll"
     "ThreadingModel"="Apartment"
```

As in earlier examples, substitute the proper GUID for `<GUID>` in the above registry file fragment.

It is possible to register a context menu handler for all files or for a single file extension. For CppExt, the .cpp file extension was used, as Listing 17.18 shows.

Listing 17.18. Registry file entries required for a context-menu handler.

```
[HKEY_CLASSES_ROOT\.cpp]
   @="cpp_auto_file"
[HKEY_CLASSES_ROOT\cpp_auto_file]
   @="C++ File"
[HKEY_CLASSES_ROOT\cpp_auto_file\shellex\ContextMenuHandlers]
   @="CppLC"
[HKEY_CLASSES_ROOT\cpp_auto_file\shellex\ContextMenuHandlers\CppLC]
   @="{d3d834c0-9a3b-11cf-82a0-00608cca2a2a}"
```

In addition, when registering a shell extension for Windows NT, you must add the CLSID for the shell extension under the following key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Shell
Extensions\Approved
```

The CppExt.reg Registry file provided on the accompanying CD-ROM contains all the entries required to register a context menu shell extension on Windows NT. To merge this file with your current Registry, right-click on the file in the Explorer and choose Merge from the context menu.

Debugging a Shell Extension

Debugging an extension to the Windows NT shell requires a few steps you might not be accustomed to using. You restart your machine, close the shell, and reload the shell into the Developer Studio debugger. The steps involved are similar to the steps used to debug any DLL:

1. Close all running applications and folders.
2. Restart Windows NT.
3. Open Visual C++ and ensure that it is the only open application.
4. Close the shell by following the usual steps to shut down Windows NT. This time, however, click the No button in the confirmation dialog box while pressing Ctrl+Alt+Shift.

27

5. After the shell shuts down, choose Setting from the Build menu. A dialog box appears.
6. Click on the Debug tab, and enter the path to the Windows NT Explorer in the Executable for Debug Session edit control. The path usually is something like `WINNT\EXPLORER.EXE`.
7. Close the dialog box.

After completing these steps, start a debug session. The shell restarts and runs inside the Developer Studio debugger. You will be able to set breakpoints and step through the code in your shell extension.

Summary

This chapter presented an overview of OLE and DDE concepts, and provided examples showing how you can use OLE and DDE to interact with the Windows NT shell. OLE is definitely the wave of the future—you should use DDE only for existing applications or to provide backward compatibility. Use OLE whenever possible for all your new projects.

The next chapter, "OLE Drag and Drop," discusses uniform data transfer, the OLE Clipboard, and moving data between applications using drag and drop.

28

Peter Norton's®

Complete Guide to Windows NT® Workstation 4

1999 Edition

Peter Norton
Richard Mansfield
John Paul Mueller

SAMS
A Division of Macmillan Computer Publishing
201 West 103rd Street, Indianapolis, Indiana 46290 USA

29

30

another category. The difference between the two is when Windows NT uses the entries in one category versus another.

Categories are the five main keys under the `My Computer` key. Categories divide the Registry into five main areas:

- `HKEY_CLASSES_ROOT`
- `HKEY_CURRENT_USER`
- `HKEY_LOCAL_MACHINE`
- `HKEY_USERS`
- `HKEY_CURRENT_CONFIG`

Each category contains a specific type of information. And although Windows NT still has to use the infamous SYSTEM.INI and WIN.INI files for antiquated applications, the use of the Registry for all other purposes does reduce the user's workload. Eventually, all applications will use the Registry to store their configuration data. The next few sections of this chapter describe the primary Registry sections (the categories) in detail.

Technical Note: You might wonder, given that installing new applications can add large amounts of data to your Registry, how big can it get? And can you define its size? The answers are: very big and yes. The maximum size is 80% of the paged pool (which itself has an upper limit of 128MB). (A *paged pool* is a zone in RAM that, when not needed, is saved to the hard drive. The pool contains system information.)

Therefore the Registry can be 102MB large. By default, the maximum Registry size will be set to 25% of the paged pool. You can specify a different maximum Registry size by locating this key: `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control` and adding a new DWORD name: **`RegistrySizeLimit`**. Double-click this new name and, for its data, type in whatever size in megabytes you want to specify.

Windows NT, however, will notice if the Registry is threatening to overflow and will warn you with a dialog box. It suggests you might want to increase the maximum in this situation.

A second way to specify the maximum Registry size is to click Start|Settings|Control Panel, and then double-click the System icon and click the Performance tab, and then click the Change button. At the bottom of this dialog box you will see the current Registry size and the maximum size. You can edit the maximum size.

You should realize, however, that just specifying a maximum size doesn't ensure that the Registry will ever be allowed to grow to that size. Also note that the *minimum* Registry size is 4MB.

31

HKEY_CLASSES_ROOT

The `HKEY_CLASSES_ROOT` category has two distinct types of entries. The first key type (remember, a key is a Regedit topic) is a file extension. Think of all the three-letter extensions you have used, such as DOC and TXT. Windows NT still uses them to differentiate one file type from another. It also uses them to associate that file type with a specific action or set of actions. Although you can't do anything with a file that uses the DLL extension, for example, it appears in this list because Windows NT needs to associate DLLs with an executable file type. The second entry type is the association itself. The file extension entries normally associate a data file with an application or an executable file with a specific Windows NT function. Below the association key are entries for the menus you see when you right-click an entry in Explorer. It also contains keys that determine the type of icon to display and other parameters associated with a particular file type. Figure 9.3 shows the typical `HKEY_CLASSES_ROOT` organization.

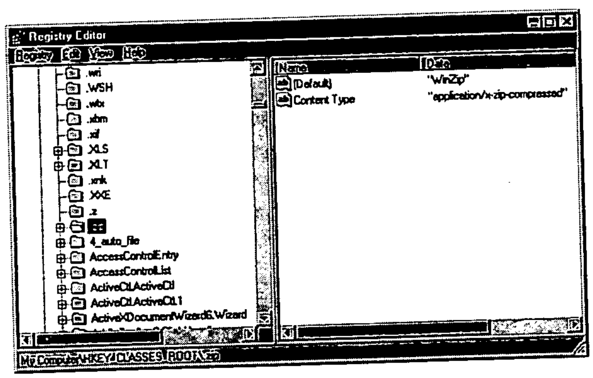

FIGURE 9.3. *A typical HKEY_CLASSES_ROOT display. Notice the distinct difference between file extension and file association keys.*

Don't let the deceptively simple appearance of this category fool you; it is one of the truly well-designed areas of the Registry. Let's take an in-depth look at the two more obvious key types. The following sections describe these entries in detail.

> **Tip:** You should always modify your application file entries to make your working environment as efficient as possible. You should never change an executable file association, however. Changing an executable file extension such as DLL could make it hard for Windows NT to start your applications, or could even crash the system. The section titled "Modifying File Associations," later in this chapter, looks at the procedure for changing an application file association.

Special Extension Subkeys

Some file extensions such as TXT provide a `ShellX` subkey. In the case of TXT and DOC, the standard subkey is `ShellNew` (the most common key). The term `ShellX` means shell extension. I like to think of it as an automated method of extending the

functionality of Windows as a whole. When you right-click the desktop and choose the New option from the context menu, all the types of files you can create are the result of shell extensions. Although `ShellNew` is the most common type of shell extension, a variety of other shell extensions are available. The actual number is limited only by your application vendor. Microsoft Excel provides no less than three different shell extension entries for the XLS file extension, for example.

A shell extension is an OLE (see Chapter 14, "Docucentricity and ActiveX") hook into Windows NT. Only an application that supports OLE 2 extensions can place a `ShellX` key into the Registry; don't add one of these keys on your own. When you see this key, you know that the application provides some type of extended OLE-related functionality. If you double-click a shortcut to a data file that no longer exists, for example, an application with the `ShellNew` shell extension asks whether you want to create a new file of the same type. (If you look at the values associated with `ShellNew`, there's always a `NullFile` entry that tells the shell extension what type of file to create.)

> **Tip:** Sometimes a file type won't appear on the context menu even though the application provides support for it. Most of the time, it happens with 16-bit applications that don't install correctly. You can use the shell extension behavior to create new files as if the context menu entry did exist. All you need to do is create a temporary file, place a shortcut to it on your desktop, and erase the temporary file. (Make certain that the application provides a `ShellNew` shell extension before you do this.) Whenever you double-click the shortcut, your application creates a new file for you. This behavior also works if you place the file shortcut in the Start Menu folder.

The Registry also has other, more generic, shell extensions. The * extension has a generic `Shellex` subkey, for example. Below this, you will see a `PropertySheetHandler` key and an all-digit key that looks like some kind of secret code: `{1F2E5C40-9550-11CE-99D2-00AA006E086C}`. Actually, the key *is* a secret code. It is a reference identifier for the DLL (a type of application) that takes care of the * extension. As you can see, shell extensions are a powerful addition to the Windows NT Registry. You should never change or delete shell extensions; let each application take care of them for you. OLE1 and 2 are, like Windows NT 3.x, essentially ancient references.

HKEY_CURRENT_USER

The HKEY_CURRENT_USER category contains a lot of "soft" settings for your machine. These soft settings tell how to configure the desktop and the keyboard. It also contains color settings and the configuration of the Start menu. All user-specific settings appear in this category, as well as settings for installed applications.

The HKEY_CURRENT_USER category is an alias of a corresponding section in the HKEY_USERS area of the Registry database. When a user logs off, Windows NT saves all

33